ICCP Coding Manual

Connor Glosser, Jos Seldenthuis, Chris Verzijl,
Erin McGarrity, and Jos Thijssen

January 22, 2014

Contents

Introduction

"There are only two kinds of programming languages:
those people always bitch about and those nobody uses."

—Bjarne Stroustrup

These coding notes are a short introduction to scientific programming, combined with a list of best practices. Programming is an art; it takes time to become experienced, but these guidelines will help you to avoid common mistakes. Follow them! They were born out of years of experience, and will save you and your instructors a lot of time. You will be spending most of your time thinking about your algorithms and debugging your code, not actually writing it. Writing clean code, following best practices, and not trying to be overly clever will make your life a lot easier.

Choosing a programming language is about picking the right tool for the job. In the case of ICCP, that means Fortran. We won't force you to use a particular language, but for most of the programs you'll write, dynamic languages such as MATLAB or Python are too slow, while C, C++, and Java lack the built-in numerical facilities[1] (with Java also being slow). Fortran is over half a century old—it first appeared in 1957—and can be a bit quirky, but for better or worse, it's still the industry standard when it comes to scientific computing. Besides, modern Fortran—the most recent standard updates appeared in 2003 and 2008—is not nearly the monstrosity it used to be; it's actually a quite pleasant language for computational physics. Even if you already have some experience with another language, take it from us that both you and your code will be faster if you use Fortran for ICCP. If you want to continue in computational physics you'll encounter it sooner or later anyway (think LAPACK), and ICCP is the perfect time to learn.

These notes are by no means a Fortran reference. High-quality documentation can be easily found online. Good starting points are `http://fortran90.org` and `http://en.wikipedia.org/wiki/Fortran_language_features`. The *Intel Fortran Compiler Language Reference* is freely available (Google it). We also recommend *Modern Fortran Explained* by Metcalf, Reid and Cohen, or *Fortran 95/2003 for Scientists & Engineers* by Chapman.

Finally, you can find an up-to-date version of all the examples shown here (as well as useful helper modules and visualizations) at `https://Github.com/cglosser/libICCP`. See the Revision control chapter for details on cloning the code from Github.

[1] It's called Fortran—FORmula TRANslation—for a reason!

Chapter 1

Getting your toes wet with Fortran and Linux

> *"1957: John Backus and IBM create FORTRAN. There's nothing funny about IBM or FORTRAN. It is a syntax error to write FORTRAN while not wearing a blue tie."*
>
> —James Iry

For this course we recommend using the GNU Fortran compiler, as it is free and of reasonably high quality.[1] Linux is the platform of choice, since it's most suited to programming and high-performance computing.[2] OS X, being a Unix variant, is also an option, but installing the required packages is generally a bit more work. In these notes we will assume that you're using Ubuntu.[3]

After booting Ubuntu, you need to install a Fortran compiler. Open a terminal and at the prompt (the $) type

```
$ sudo aptitude install gfortran gfortran-doc
```

Note that anything you type in the console is case-sensitive! This command grants `aptitude` privileges to search for and then install the `gfortran` package and its documentation. You may also find it useful to install the LaTeX typesetting system for writing reports. You can install it with another call to `aptitude`:

```
$ sudo aptitude install texlive # or texlive-full for a complete installation
```

Once you have `gfortran` installed, you can start writing programs. You will generally compile and run your programs from a terminal window (also called a *console*). A few useful commands include:

[1] You're welcome to use the Intel Fortran compiler, which is free on Linux, but remember to change the compiler flags, since they differ from `gfortran`.

[2] As of this writing, 476 of the TOP500 supercomputers in the world run some form of Linux.

[3] You can use Ubuntu as a native installation, run it from a flash drive, under a virtual machine, or under Windows using Wubi. Ask your instructor.

```
$ ls              # display the contents of the current directory
$ ls -l           # display contents with extra details
$ cp file path    # copy 'file' to 'path'
$ mv file path    # move 'file' to 'path'
$ rm file         # remove 'file'
$ mkdir dir       # create a new directory called 'dir'
$ cd dir          # change current directory to 'dir'
$ rmdir dir       # remove directory 'dir' (only if it's empty)
$ rm -r dir       # remove directory 'dir' (even if it's not empty)
$ man cmd         # provide documentation on 'cmd'
```

For your first program, open a terminal, create a directory for ICCP files and open your first Fortran file by typing

```
$ mkdir iccp          # create a new directory called 'iccp'
$ cd iccp             # move to the new directory
$ gedit myprog.f90    # open a text editor with the file 'myprog.f90'
```

The gedit text editor pops open in which you can type the program in Listing 1.1. You can probably guess what this program does, however, a few remarks are in order:

- The program starts with a declaration of variables. `real(8)` denotes a floating-point variable with double (8 byte) precision. Similarly, `integer` denotes an integer number.[4] Not specifying a size generally defaults to 4-byte precision. `implicit none` prevents Fortran from trying to infer the type from the variable name, which is a major source of bugs—*always include this!*
- The attribute `parameter` specifies that we are declaring a constant. Although Fortran is case-insensitive, it is considered good practice to always use uppercase names for constant variables.
- Note the calculation of π as `4*atan(1d0)`—convenient and accurate.
- Single-precision (4 byte) floating-point numbers can be written as `0.1` or `1e-1` (scientific notation). For double-precision numbers, use `d` instead of `e` in scientific notation, *e.g.*, `1d-1`. It is also possible to specify the precision as a suffix: `1.0_4` for single and `1.0_8` for double precision. This also works for integers.
- `integer :: fibonacci(N)` allocates an array of 20 integers, with the array index[5] running from 1 to 20.
- '$*$' is multiplication, '$**$' is exponentiation.
- The `print` statement on line 24 contains the formatting string `"(4I6)"`. This tells Fortran to print 4 records of integer type per line, each taking up 6 characters. Format strings for other datatypes include E$w.d$ (real – decimal form), ES$w.d$ (real – scientific form), EN$w.d$

[4] Fortran intrinsic types include `logical`, `integer`, `real`, `complex`, and `character` data.
[5] Array indices can start at any integer by replacing `N` with a lower and upper bound separated with a colon.

(real – engineering form), Lw (logical), and A (characters). Here, w gives the width of a record and d gives the number of places right of the decimal.

- Comments in Fortran start with '`!`' and last until the end of the line.
- Dividing integers **results in an integer**, so `3 / 2 == 1` instead of 1.5 as you might expect. Multiplying by `1d0` on line 27 forces Fortran to do a double-precision floating-point calculation.

Now we compile and run the program. Compiling means translating the Fortran source code into machine code (processor instructions). This can be done by simply typing

```
$ gfortran myprog.f90
```

which will result in the executable file `a.out`. This is the default name for the output. You can specify a different name by compiling with

```
$ gfortran myprog.f90 -o myprog
```

which results in a program with the name `myprog`. During compilation, the compiler may generate error messages and warnings. The errors must be fixed before an actual running program is produced, and it is good practice to also make sure no warnings are generated.

The compilation process can be tuned by passing certain command-line arguments to the compiler. We recommend using always at least the following:

```
$ gfortran -Wall -Wextra -march=native -O3 myprog.f90 -o myprog
```

- `-Wall` and `-Wextra` turn on all warnings. This may generate a lot of messages, but fixing them all leads to much cleaner code. This can be a huge time saver; not only for you, but also for your instructors! If your program doesn't behave as expected, first try to fix all warnings before asking your instructors for help.
- `-march=native` tells the compiler to generate machine code using all available processor instructions on your machine. On modern CPUs this leads to much faster code, since `gfortran` can use vector instructions. The downside is that it can result in executables that will not run on a different machine.
- `-O3` turns on all optimizations. This increases the compilation time significantly, although it should still be fast enough for the programs you'll write in ICCP. The runtime of your program, on the other hand, will dramatically decrease. The only reason not to use this flag is that it might interfere with the debugger (see below).
- A possible additional optimization flag is `-ffast-math`. This flag enables floating-point optimizations which might reduce accuracy or result in incorrect behavior, especially in situations such as divide-by-zero. We therefore recommend not using this flag until you have verified that your program produces the correct results. After that you can use it to make your program faster, but do check that it still behaves correctly.

- Finally, if your code starts throwing segmentation faults (segfaults), consider using the `-fbounds-check` flag. Segfaults usually occur because you've run an index off the end of an array, so compiling with this flag turns on runtime bounds checking (at the expense of significantly reduced performance).

If the program compiled correctly, you can run it by typing `./a.out` or `./myprog`. Note that `./` specifies the *path, i.e.,* the location where to find the program. The dot means the current directory (similarly, `..` refers to the parent directory), and the slash separates directories and file names (like the backslash in DOS and Windows).

Listing 1.1: examples/myProg.f90

```
program MyProg

    implicit none

    real(8), parameter :: PI = 4 * atan(1d0)
    integer, parameter :: N = 20

    real(8) :: radius, surface, circumference
    integer :: i, fibonacci(N)

    print *, "Give the radius"
    read *, radius
    surface = PI * radius**2
    circumference = 2 * PI * radius
    print *, "The surface of the circle is ", surface
    print *, "and the circumference is     ", circumference

    fibonacci(1) = 0
    fibonacci(2) = 1
    do i = 3, N
        fibonacci(i) = fibonacci(i - 1) + fibonacci(i - 2)
    end do
    print *, "Fibonacci sequence:"
    print "(4I6)", fibonacci
    ! The ratio of neighboring Fibonacci numbers converges to the
    ! golden ratio.
    print *, "Golden ratio:", 1d0 * fibonacci(N) / fibonacci(N - 1)

end program
```

Chapter 2

Structuring your code

(to keep your instructors sane!)

> *"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."*
> . . . because that guy might be you in six months.
>
> —Martin Golding

Readability is the most important quality of your code. It's more important even than being bug-free; you can fix a readable but incorrect program, while an unreadable program is next to useless even if it works. As an example of what not to do, take a look at the gem in Listing 2.1. Originally written in FORTRAN IV for an IBM 1130 (in 1969!), it is a perfect example of spaghetti code and it makes me want to pick out my eyes with a fondue fork. If you can figure out what it does, I'll buy you a beer.[1]

To keep your code readable, you need to structure your program in a smart way while following a reasonable coding style. A coding style encompasses things like naming conventions, indentation, comment rules, *etc.*; see the Coding style chapter below for our recommendations. In this chapter we focus on structure.

2.1 Functions and subroutines

As a (silly) example, let's say you need to sum all integers in a certain interval. Programmers are lazy, so we're going to write a program to do that for us. Gauss, of course, found a way to do this very quickly, but for the sake of the example, let's pretend we don't know about it and use Listing 2.2 instead. Note that we can't use the word `sum` as a variable name, since that's a Fortran intrinsic, so we use `total` instead. This program is simple, clean, and doesn't need comments to show what it does. However, we can do better. Summing numbers in an interval is a generic mathematical operation, which we might need to do more often. It therefore makes sense to move that code to a separate subroutine which we only have to write once and can

[1] It's part of one of the first open source programs in history, so no snooping with Google!

then reuse as many times as we need. We can do this in Fortran, with the subroutine seen in Listing 2.3. Fortran subroutines are invoked with the keyword **call** followed by the name of the routine and an optional list of parameters in parentheses. The code for the subroutine itself appears after the keyword **contains**. We don't have to type **implicit none** again, since that's inherited from the program. When declaring the parameters, we need to specify not only the type, but also how we're going to use them, *i.e.*, the *intent*. For input and output parameters, we use **intent(in)** and **intent(out)**, respectively. For parameters functioning as both, we use **intent(inout)**.

`stupid_sum` is essentially just a mathematical function. Wouldn't it then be nice to be able to use it that way, by simply invoking it with `total = stupid_sum(first, last)`? Fortunately, Listing 2.4 demonstrates a way to do just that. A function in Fortran is just a subroutine returning a value. We therefore need to specify the type of (the result of) the function as well as its arguments. This type is given before the keyword **function**. **result**`(total)` tells Fortran that the local variable `total` (of type **integer**) holds the return value of the function.

Splitting reusable code off into subroutines and functions is absolutely necessary for any non-trivial program. Always take the time to think about the structure of your program before you start typing. The golden rule here is: *every function or subroutine should do one thing and do it well.* In practice this also means they should be small; if your function doesn't fit on a single screen, that's usually an indication that you should split it up. Besides making the code more readable, this also makes it easier to modify. Let's say that you suddenly discover Gauss' trick for summing intervals. All you need to do then is replace `stupid_sum` by `smart_sum` (Listing 2.5). You don't have to search through all your code to change the calculation everywhere; you just update the function and your entire program suddenly becomes a lot faster.

2.2 Modules

Many functions and subroutines are reusable not only within a single program, but also between programs. In that case it is a good idea to place those routines in a module.[2] For example, if you envisage using Gauss' algorithm very often in the future, you might want to create the module `gauss.f90` given in Listing 2.6. By default, all functions and subroutines in a module are visible to external programs. This can cause collisions with similarly names functions in other modules, especially in the case of helper routines that are only used within the module. Common practice is therefore to specify the **private** keyword to change the default behavior; only the routines that are explicitly made public are then usable elsewhere.

Any program can now use the functions in `gauss.f90` by simply specifying **use** `Gauss` (Listing 2.7). You now have two different files that need to be compiled to produce a working program. It's possible to do this by hand, but there is a much easier way with The magic of Makefiles.

[2] Some subroutines—particularly those that allocate arrays that appear as arguments—require you to write an explicit interface for them *or* place them within a module (so the compiler can generate the interface for you).

Listing 2.1: examples/snppic.f

```
      SUBROUTINE SNPPIC
      DIMENSION ILINE(133),INUM(50),ICHR(50)
      COMMON ISET
      DATA IBLNK/1H /
      DO 4 I=1,133
    4 ILINE(I)=IBLNK
    5 K=1
   10 READ (2,1000) (INUM(I),ICHR(I),I=1,ISET)
      DO 40 I=1,ISET
      IF (INUM(I) .NE. -1) GOTO 100
      DO 15 L=K,133
   15 ILINE(L)=ICHR(I)
      WRITE (7,2000) (ILINE(K),K=1,133)
      ILINE(1)=IBLNK
      DO 20 K=2,133
   20 ILINE(K)=ICHR(I)
      K=1
  100 IF (INUM(I) .EQ. -2) GOTO 200
      IF (INUM(I) .EQ. 0) GOTO 40
      DO 30 J=1,INUM(I)
      ILINE(K)=ICHR(I)
      K=K+1
   30 CONTINUE
   40 CONTINUE
      GOTO 10
  200 RETURN
 1000 FORMAT (25(I2,A1))
 2000 FORMAT (133A1)
      END
```

Listing 2.2: examples/SumsBeforeGauss.f90

```
program SumsBeforeGauss

    implicit none

    integer :: first, last, total, i

    print *, "First:"
    read *, first
    print *, "Last:"
    read *, last

    total = 0
    do i = first, last
        total = total + i
    end do

    print *, "Total:", total

end program
```

Listing 2.3: examples/SumsBeforeGauss_subroutine.f90

```
program SumsBeforeGauss

    implicit none

    integer :: first, last, total

    print *, "First:"
    read *, first
    print *, "Last:"
    read *, last

    call stupid_sum(first, last, total)

    print *, "Total:", total

contains

    subroutine stupid_sum(a, b, total)

        integer, intent(in) :: a, b
        integer, intent(out) :: total

        integer :: i

        total = 0
        do i = a, b
            total = total + i
        end do

    end subroutine

end program
```

Listing 2.4: examples/SumsBeforeGauss_function.f90

```
program SumsBeforeGauss

    implicit none

    integer :: first, last

    print *, "First:"
    read *, first
    print *, "Last:"
    read *, last

    print *, "Total:", stupid_sum(first, last)

contains

    integer function stupid_sum(a, b) result(total)

        integer, intent(in) :: a, b

        integer :: i

        total = 0
        do i = a, b
            total = total + i
        end do

    end function

end program
```

Listing 2.5: examples/smart_sum_function.f90

```
! Gauss was a genius! 1 + 2 + 3 + ... + n = n * (n + 1) / 2
integer function smart_sum(a, b) result(total)

    integer, intent(in) :: a, b

    total = (b * (b + 1)) / 2 - (a * (a - 1)) / 2

end function
```

Listing 2.6: examples/gauss.f90

```
module Gauss

    implicit none
    private

    public smart_sum

contains

    integer function smart_sum(a, b) result(total)

        integer, intent(in) :: a, b

        total = (b * (b + 1)) / 2 - (a * (a - 1)) / 2

    end function

end module
```

Listing 2.7: examples/Sums.f90

```
program Sums

    use Gauss

    implicit none

    integer :: first, last

    print *, "First:"
    read *, first
    print *, "Last:"
    read *, last

    print *, "Total:", smart_sum(first, last)

end program
```

Chapter 3

The magic of Makefiles

Once your programs become larger, it is convenient to split the source code into several files. However, to obtain a running program, each of these files has to be compiled separately (resulting in a *relocatable object file*, with extension `.o`) and then combined (linked) into a single program. Doing this by hand quickly becomes tedious, so we resort to using Makefiles. We create a file called `Makefile` that contains all compiler invocations, and then we can compile the program by simply typing

```
$ make
```

Similarly, we can remove all compiler-generated files with

```
$ make clean
```

The `make` program is smart enough to only compile files that have changed; so if you just fixed a bug in `file6.f90`, it will only recompile that file (generating `file6.o`) and perform the linking step, without touching the other files.

So, what does a `Makefile` look like? As an example, let's go through Listing 3.1 step by step.

- The first lines declare a few variables for later use. `FC` is set to the Fortran compiler `gfortran`, `FFLAGS` contains the compiler flags and `LDFLAGS` the linker flags. The `COMPILE` and `LINK` variables combine these to get the compile and link commands.
- Most non-trivial programs make use of several function libraries, such as the Linear Algebra PACKage (LAPACK). These libraries are stored under `/usr/lib` and have filenames such as `liblapack.a` or `liblapack.so`. You can link such a library with your program by specifying the `-llapack` argument to the linker. Note that you have to omit both `lib` and the extension (`.a` or `.so`) from the library name. Since libraries have to appear after the object files for the linker, they are specified in the separate variable `LIBS`. If the library cannot be found in the system paths, you can tell the compiler to search for it in the directory *`PATH`* by adding the option `-L`*`PATH`* to `LDFLAGS`, *e.g.*, `LDFLAGS = -L$HOME/mylib`.
- Makefiles are built around rules, which have the form '*`target`*`:  `*`dependencies`*' followed by some lines with the commands needed to create the target from the dependencies.

Listing 3.1: examples/example_makefile

```
1  FC = gfortran
2  FFLAGS = -Wall -Wextra -march=native -O3
3  LDFLAGS =
4  LIBS = -llapack
5
6  COMPILE = $(FC) $(FFLAGS)
7  LINK = $(FC) $(LDFLAGS)
8
9  OBJS =
10 OBJS += mymod.o
11 OBJS += extra.o
12 OBJS += myprog.o
13
14 all: myprog
15
16 myprog: $(OBJS)
17 ␣␣␣␣␣␣␣␣$(LINK) -o $@ $^ $(LIBS)
18
19 %.o: %.f90
20 ␣␣␣␣␣␣␣␣$(COMPILE) -o $@ -c $<
21
22 .PHONY: clean
23 clean:
24 ␣␣␣␣␣␣␣␣$(RM) myprog $(OBJS) *.mod
```

Except for a few special targets, such as `all` and `clean`, `make` will assume the target is a file, and try to recreate it if the target is older than the dependencies. *Note that the lines with the build commands must start with a tab character, not spaces!*

- If `make` is invoked without any parameters, the default target is the first entry in the `Makefile` (in this case `all`), which depends on `myprog` here. `myprog` in turn depends on the list of object files in `$(OBJS)`. This is the reason we only specified the object files and not the source code files. The executable is generated by invoking the linker, which combines the object files with the external libraries. In the `Makefile`, `$@` is an alias for the target. On line 17, for example, `$@` is replaced by `myprog` when running `make`. Similarly, `$^` is an alias for all dependencies, in this case the list `$(OBJS)`.
- Since we don't want to specify a separate rule for every object file, we use *wildcards*. `%.o: %.f90` means every object file ending in `.o` depends on the corresponding source file, ending in `.f90`. `$<` on the command line is an alias for the first prerequisite, in this case the source file. The `-c` option tell the compiler to only perform the compilation step to create an object file, and to skip linking. Note that the object files are compiled in the

order specified in `$(OBJS)`. This means that if the program file `myprog` depends on the module `mymod`, `mymod.o` needs to be listed before `myprog.o`.

- The special target `clean` is used to delete all compiler-generated files. This includes the executable `myprog`, the object files listed in `$(OBJS)` and any automatically generated module files (with extension `.mod`).

In the rest of these notes we will assume that you have created a `Makefile` for each project based on this template.

Chapter 4

Debugging

(or, where you'll spend 90% of your time)

> *"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"*
>
> *"The most effective debugging tool is still careful thought, coupled with judiciously placed print statements."*
>
> —Brian Kernighan

The `MyProg` program does not contain any bugs (fingers crossed) and works as expected. However, any non-trivial program will be buggy after your first attempt at writing it. *In this course you will spend the majority of your time debugging your algorithms and your code, not on the actual writing!* To ease your pain, we recommend following the golden rules of debugging:

- **Always fix the first error first.** Errors have a tendency to cascade, with a simple typo early on generating loads of errors in the rest of your program. If you find yourself with a ton of unmanageable issues, use the `-Wfatal-errors` flag to halt compilation after the first error so you can fix them individually.
- **Don't ignore warnings.** Warnings are often an indication that your code doesn't actually do what you think it does. And if it does happen to work, it's an indication of bad programming practice, or worse, style. You can use the `-Werror` flag to elevate warnings into errors, forcing you to fix them before the compiler will produce an executable.
- **Test early, test often.** No matter how large the project, always start with a trivial program and make sure that works. Then add functionality bit by bit, each time making sure it compiles and works.
- **Produce lots of output.** You might think that there can't possibly be a bug in that beautiful expression, but you're wrong. There are countless innocuous subtleties to

programming that can result in inaccuracies or unintended side effects. Test your program by printing the intermediate results.

- **Keep a rubber ducky on your desk.** You've been staring at your code for hours, it's obviously correct, yet it doesn't work. In that case the best thing to do is to try and explain your code to someone else. It doesn't matter who it is; they don't have to understand Fortran, or even programming in general. It can be your partner, your mother, your dog, or even a rubber ducky. Just explain it line-by-line; ten to one says you'll find the bug.

Even if you do follow these rules—*and you should!*—you will run into bugs that you don't understand. Often they can be fixed by backtracking and checking every intermediate step, but bugs can be elusive. In those cases a debugger comes in handy. To use a debugger under Ubuntu, install the `gdb` and `ddd` packages. You'll also need to compile your program with the `-g` flag to generate debug symbols, and turn off any optimizations as they might reorder expressions in your code, *e.g.*,

```
$ gfortran -Wall -Wextra -march=native -g myprog.f90 -o myprog
```

If the program compiles, you can invoke the debugger with

```
$ ddd ./myprog
```

This will open a window showing the source code of your program and a GDB console (the actual debugger). You can run the program by pressing the 'Run' button, and interrupt a running program with 'Interrupt'. You can also insert so-called breakpoints into your program by right-clicking on a line and selecting 'Set Breakpoint'. If you then run your program, it will execute until it encounters the breakpoint, where it will pause. You can then step through your code by clicking 'Step' or 'Next'. 'Step' will move to the next statement in your code. 'Next' does the same, but it will step over subroutines instead of entering them. You can inspect variables by right-clicking on the name and selecting 'Print x' or 'Display x'.

Play around with the debugger for a while until you're comfortable using it. It will come in very handy when you're hunting bugs, and as we said before, that's what you'll be spending most of your time on.

Chapter 5

Optimization

(or, where you *shouldn't* spend 90% of your time)

> *"Premature optimization is the root of all evil."*
>
> —Donald E. Knuth

> *"Rules of Optimization:*
> *Rule 1: Don't do it.*
> *Rule 2 (for experts only): Don't do it yet."*
>
> —Michael A. Jackson

Impatience is the most important characteristic of a good programmer. However, this does result in a tendency to focus too much or too early on optimizing code. Don't. Always make sure your code is correct and readable before you try to make it faster. And when you do try to make it faster, conventional wisdom says: *don't optimize your code, optimize your algorithms.* This may seen counter-intuitive; after all, what's wrong with fast code? However, modern CPUs are complex beasts, and unless you're an expert, it's not at all obvious which expressions result in the fastest code. This also means that a lot of tricks that used to be valid ten years ago no longer are. Let's look at a few examples:

- **Division is slower than multiplication by an inverse.** Although true, optimizing for this results in code that is hard to read, and possibly less accurate. Besides, when you're using the `-ffast-math` option, `gfortran` will do this for you. A similar observation holds for exponentiation; it is slower than multiplication in general, but `gfortran` will automatically convert `x**2` to `x * x`, provided the exponent is **`integer`**.
- **Unwinding small loops reduces overhead.** Common wisdom says that

```
do i = 1, 3
    call do_something(i)
end do
```

  is slower than

```
call do_something(1)
call do_something(2)
call do_something(3)
```

because of the loop's overhead. Although again true in principle, modern CPUs use branch prediction to dramatically reduce this overhead. Moreover, unwinding loops results in more machine code, which might actually hurt performance if it no longer fits in the CPU instruction cache. When you compile with `-march=native`, the compiler knows about the CPU limitations and will decide whether loop unwinding is the smart thing to do.

- **Calculate everything only once.** Let's say that x^2 occurs four times in your expression, then it makes sense to calculate `x2 = x**2` separately and use the result four times, right? Possibly yes, but only if it improves readability. Common subexpression elimination is something compilers have done for ages, so again `gfortran` will do this for you.

There are countless more examples, but I think you get the idea. In short: optimize for readability; write clean code and let the compiler worry about the rest. Unless you're intimately familiar with things like cache misses, branch prediction and vector instructions, you won't be able to outsmart the compiler.

5.1 Some optimizations you are allowed to use

That being said, there are a few general rules that will prevent your code from accidentally becoming horrendously slow:

- **Don't reinvent the wheel.** Don't try to write your own matrix multiplication routine, use `matmul`, or the routines from BLAS (see the Linear algebra appendix). Fortran has many built-in numerical functions that are much faster than anything you'll be able to write—use them! (Google 'Fortran intrinsics' to get an overview.)

- **Use array operations.** You can add two arrays by writing

```
do i = 1, n
    do j = 1, m
        c(i, j) = a(i, j) + b(i, j)
    end do
end do
```

or simply by writing `c = a + b`. Guess which is both faster and more readable?

- **Keep inner loops small.** The code inside loops, especially in nested loops, is executed many times. This code will likely be the bottleneck of your program, so it makes sense to concentrate your optimization efforts there. Move as much code out of the inner loop as possible.

- **Don't use system calls in nested loops.** Certain operations, such as allocating memory, or writing to the screen or disk, require a system call—a function call to the underlying operating system. Such calls are expensive, so try to avoid them in nested loops. Especially disk I/O is slow. Luckily, modern operating systems are smart enough to buffer these operations, but they can still become a bottleneck.

- **Be smart about memory usage.** First, don't use more memory than you need, especially more memory than is physically available. That will result in paging (storing and retrieving data from disk), which is orders of magnitude slower than direct memory access. Second, think about memory layout. Fortran uses column-major ordering for arrays.[1] This means that the array

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix}$$

  is actually stored in memory as $[a, e, i, b, f, j, c, g, k, d, h, l]$. Because of this, operations within a single column are much faster than operations within a single row; the processor can easily load a whole column into the cache, but not a whole row. It is therefore important to carefully consider your array layout. For example, if you have an array containing the coordinates of many particles, then a $3 \times n$ layout will be more efficient than $n \times 3$. In the former, the x, y and z coordinates are close together in memory, meaning that operations on single particles will be fast.

These optimizations are more about preventing your code from becoming unnecessarily slow, than trying to squeeze the last drop of performance out of it. Besides, low-level optimizations generally yield only about a factor of 2 speedup, and that's if you know what you're doing. To see real speedups, you're better off profiling your code to determine bottlenecks and optimizing your algorithms by improving scalability.

5.2 Scalability; or, the importance of big O

If you want your program to be fast, choose your algorithms carefully. The fact that you can approximate an integral by a simple sum does not mean that's the best way to do it. In fact, the naïve approach is often the worst method, both in terms of computational complexity, *i.e.*, how much more expensive the calculation becomes when you increase the size of the problem, and numerical accuracy, *i.e.*, how small the error becomes when you increase the number of steps/iterations. Both the complexity and the accuracy of an algorithm are generally quantified using *big O* notation. Big O notation is essentially a Taylor expansion of your algorithm; usually in the step size h to approximate the error, or the dimension n to approximate the computation time. In general, only the highest or lowest power matters, since that will be the dominant term for large n or small h. Different algorithms for solving the same problem are compared by looking at their big O characteristics. However, scaling is not everything. Some algorithms, such as the famous Coppersmith-Winograd algorithm for matrix multiplication, scale very well,

[1] MATLAB and R adopt the same convention as Fortran; C, Python, and Mathematica use row-major ordering.

but have such a large prefactor in their computational cost, that they're inefficient for all but the largest problems.

Long story short; spend some time doing research before you start coding. Look at different algorithms, compare their accuracies and complexities—don't forget the complexity of implementation!—and only then start writing. Don't just go for the seemingly obvious approach. For example, naïvely integrating an ordinary differential equation (ODE), *e.g.*, using the Euler method, will make an error of order $\mathcal{O}(h^2)$ at each integration step, giving a total error of $\mathcal{O}(h)$. Moreover, this method is unstable, meaning that for stiff equations the solution will oscillate wildly and the error grow very large. The Runge-Kutta method, on the other hand, makes an error of $\mathcal{O}(h^5)$ per step, giving a total error of $\mathcal{O}(h^4)$ (which is why it's generally referred to as 'RK4'). Additionally, the method is numerically stable. Many such methods are significantly more accurate than the Euler method, without being harder to implement. Furthermore, better scaling of the error means that they need fewer iterations, making them also much faster; it's a win-win.

Numerical analysis is a large field, and it can be quite daunting trying to understand the merits of different methods, or even finding them. For this, we recommend the book *Numerical Recipes*[2]. It covers a wide range of topics, focuses on best practices, and does not ignore coding effort when comparing algorithms. It even includes the full source code of the implementations! We recommend always using it as your starting point, with one exception: linear algebra (see Appendix A).

We conclude this section with an overview of the computational complexity of certain common operations. This will give you an idea of the most likely bottleneck in your code.

- All scalar operations are $\mathcal{O}(1)$, *i.e.*, constant time. Integer operations are much faster than floating-point operations. Addition and subtraction are the fastest, followed by multiplication and finally division. Functions like **`sqrt`**, **`exp`**, **`log`**, **`sin`**, **`cos`**, *etc.*, are not always available as single processor instructions, especially for complex numbers, in which case they are implemented as actual function calls. It goes without saying that this is much slower than basic arithmetic.

- Scaling a vector or a matrix means performing an operation on each element. The complexity is therefore $\mathcal{O}(n)$ for vectors and $\mathcal{O}(n^2)$ for matrices, where n is the dimension.

- Vector addition and taking an inner product are $\mathcal{O}(n)$, matrix addition is $\mathcal{O}(n^2)$.

- Matrix-vector multiplication means taking an inner product for each row, so the complexity is $\mathcal{O}(n^2)$.

- Solving a system of linear equations is also $\mathcal{O}(n^2)$, but the prefactor is much larger than for matrix-vector multiplication.

- Naïve matrix-matrix multiplication is $\mathcal{O}(n^3)$. Libraries such as BLAS and LAPACK are smarter and do it in $\mathcal{O}(n^{\log_2(7)}) \approx \mathcal{O}(n^{2.807})$. Another reason to avoid writing this yourself.[3]

[2] Available free online at `http://apps.nrbook.com/fortran/index.html`

[3] The Fortran intrinsic `matmul` is actually implemented using BLAS.

- Taking the inverse of a matrix and calculating the eigenvalues/eigenvectors are both $\mathcal{O}\left(n^3\right)$. However, eigendecomposition has a much larger prefactor and is therefore significantly slower. It is logically the most expensive matrix operation, since all operations on a diagonal matrix are $\mathcal{O}(n)$.
- Iterative algorithms like conjugate-gradient or Krylov-subspace methods are supra-convergent in the number of iterations, *i.e.*, the error decreases by a constant factor at each iteration. They are especially suitable for sparse matrices, since each iteration is $\mathcal{O}(n)$, where n is the number of non-zero elements instead of the dimension.

Chapter 6

Coding style

> *"Consistently separating words by spaces became a general custom about the tenth century A.D., and lasted until about 1957, when FORTRAN abandoned the practice."*
>
> —Sun FORTRAN Reference Manual

> *"Simplicity is prerequisite for reliability."*
>
> —Edsgar Dijkstra

A major part of writing clean code is following a sensible coding style. Although style is subjective and a matter of taste, consistency is not. Which style you pick is not as important as sticking to it. However, certain styles are objectively better than others. While we won't force a particular style, you will make it a lot easier for your instructors to read your code if you follow these guidelines, and that's always a good thing. Remember: you will be spending a lot more time reading your code than writing it; keep it clean.

- **Line width**—Limit the line width of your source to 80 characters. This has been standard practice for decades, and vastly improves both readability and the ability to cleanly print your code. If you do need more than 80 characters on a line, use the Fortran continuation character '&'.
- **Indentation**—Indent every block, be it a `program`, `module`, `subroutine`, `function`, `do` or `if` block by 2 or 4 spaces. *Don't use tabs!* Code that looks properly aligned on your machine may look horrible on another because of a different tab setting. Proper indentation makes it clear where control structures begin and end, and is a major boon to readability. If the indentation puts you over the 80 character limit, that's an indication that you may need to refactor your code intro smaller subroutines. Four levels of indentation should generally be the maximum.
- **Spaces**—Put spaces around binary operators, including `::`, and after keywords and commas, not after unary operators, function names or around parentheses. For example

```
real(8) :: x, y
x = func1()
if ((x < 0) .or. (x > 1)) then
    y = func2(-2, 2*x + 3)
end if
```

- **Naming**—Identifiers in Fortran can't contain spaces. Use `CamelCase` to separate words in program and module names, and `lower_case_with_underscores` for function and variable names. Names should be descriptive, but they don't have to be overly long. You can use `calc_deriv` instead of `calculate_derivative`, but naming your function `cd` is a shooting offense. Unfortunately, until the appearance of Fortran 90, only the first six characters of an identifier were significant, leading to such beautifully transparent function names as `DSYTRF`[1] in legacy codes like LAPACK. They could get away with it because it was the eighties; you can't. One exception to this rule is using short identifiers like `i`, `j` and `k` as loop counters, or `x` and `y` as arguments to mathematical functions. However, these should always be local variables or function arguments, never global identifiers.

- **Functions and subroutines**—Functions and subroutines should follow the Unix philosophy: do one thing and do it well. Split your program into small self-contained subunits. This makes it a lot easier to understand your code, and to modify it afterwards. If a function does not fit in its entirety on a single screen, it is too long—split it up. Don't worry about the overhead of calling a function. If you use optimization flags like `-O3`, the compiler will try to inline them anyway.

- **Modules**—Separate your program into logical units. Combine related functions and subroutines into modules. For example, a single module with function integrators, or ODE solvers. If you use the object-oriented features of Fortran, it is also good practice to create separate modules for classes, *e.g.*, a sparse-matrix object.

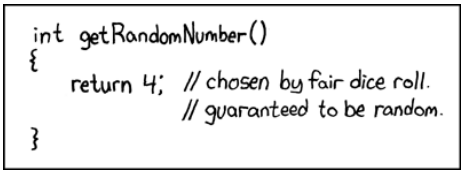
- **Comments**—Comments are kind of a double-edged sword. If used wrongly, they can actually hurt readability. The golden rule is: never explain *how* it works, just *what* it does. The how should be evident from the code itself. *"Good code is its own best documentation."* If you follow the other guidelines regarding naming and keeping functions small, you can generally suffice with a small comment at the beginning of every function or subroutine.

One final point, not of style, but important nonetheless: start every program and module with **`implicit none`**. This is an unfortunate holdover from older versions of Fortran; it forces you to explicitly declare all variables, instead of allowing Fortran to infer the type from the name. Forgetting **`implicit none`** and making a typo in one of your variable names is a major source of bugs. *Never ever omit this!*

[1] DSYTRF computes the factorization of a real symmetric matrix using the Bunch-Kaufman diagonal pivoting method, in case you wondered.

Chapter 7

A short word on random numbers



—Randall Munroe (XKCD)

When generating random numbers, you'll want to take care in the methods you use; some RNGs are most definitely better than others. You can always code your own, but Fortran has a few reasonably good subroutines to do it for you. The default `gfortran` installation provides access to a low-level **`rand`**`()` function, but you'll almost certainly want to use the more sophisticated **`random_number`** subroutine instead; it generates much higher quality random numbers and automatically threads over (fills) arrays. When you've ironed out your algorithm and want to start collecting data, you should also remember to seed the RNG. Computers generate random numbers by way of a procedural mathematical formula, so using the same seed produces the same series of random values. The subroutine in Listing 7.1 will seed the RNG with entropy collected by the system if it's available, otherwise it will use the current time & process ID.

The Fortran random number generator produces uniformly distributed reals on the interval[1] $[0, 1)$, but simulations often require numbers drawn from a different distribution. For this, we again turn you to *Numerical Recipes*; it describes several methods for generating nonuniform random variates including rejection sampling for gamma, poisson, and binomial distributions and the Box-Muller transform for normal distributions.

[1] Be careful with operations that might fail with `0.0d0`.

Listing 7.1: examples/init_random_seed.f90

```
subroutine init_random_seed()
  implicit none
  integer, allocatable :: seed(:)
  integer :: i, n, un, istat, dt(8), pid, t(2), s
  integer(8) :: count, tms

  call random_seed(size = n)
  allocate(seed(n))
  open(newunit=un, file="/dev/urandom", access="stream", &
       form="unformatted", action="read", status="old", iostat=istat)
  if (istat == 0) then
     read(un) seed
     close(un)
  else
     call system_clock(count)
     if (count /= 0) then
        t = transfer(count, t)
     else
        call date_and_time(values=dt)
        tms = (dt(1) - 1970) * 365_8 * 24 * 60 * 60 * 1000 &
             + dt(2) * 31_8 * 24 * 60 * 60 * 1000 &
             + dt(3) * 24 * 60 * 60 * 60 * 1000 &
             + dt(5) * 60 * 60 * 1000 &
             + dt(6) * 60 * 1000 + dt(7) * 1000 &
             + dt(8)
        t = transfer(tms, t)
     end if
     s = ieor(t(1), t(2))
     pid = getpid() + 1099279 ! Add a prime
     s = ieor(s, pid)
     if (n >= 3) then
        seed(1) = t(1) + 36269
        seed(2) = t(2) + 72551
        seed(3) = pid
        if (n > 3) then
           seed(4:) = s + 37 * (/ (i, i = 0, n - 4) /)
        end if
     else
        seed = s + 37 * (/ (i, i = 0, n - 1 ) /)
     end if
  end if
  call random_seed(put=seed)
end subroutine init_random_seed
```

Appendix A

Linear algebra

When it comes to linear algebra, don't try to write your own routines, always use the BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage)[1] libraries. Everyone does. Maple, Mathematica, Matlab and Python (with NumPy and SciPy) all use it under the hood. The `dot_product` and `matmul` intrinsics in Fortran are implemented with it. It's installed on every supercomputer and contains some of the most sophisticated and optimized code known to man. The fact that these libraries are written in Fortran is actually one of the main reasons to use Fortran for high-performance computing. Nevertheless, BLAS and LAPACK can be a bit low-level sometimes, and the Fortran 77 syntax doesn't help matters much. For your convenience, we therefore provide two functions for common tasks: calculating the inverse and the eigenvalues + eigenvectors of a real symmetric matrix. This should help you get started. For other functions, use Google. BLAS and LAPACK are quite extensively documented.

[1] `http://www.netlib.org/lapack/`. Under Ubuntu, install the `liblapack-dev` package to obtain BLAS and LAPACK, and don't forget to add `-lblas` and `-llapack` to `$(LIBS)` in your `Makefile`.

Listing A.1: examples/invertMatrix.f90

```
! Calculates the inverse of a real symmetric matrix.
!
! a - On input, a real symmetric matrix. Only the upper triangular part is
!     accessed. On output, A is overwritten by its inverse.
subroutine matinvrs(a)

    real(8), intent(inout) :: a(:, :)

    integer, external :: ilaenv

    integer              :: i, j, n, nb, lwork, info, ipiv(size(a, 1))
    real(8), allocatable :: work(:)

    n = size(a, 1)

    ! Calculate the optimal size of the workspace array.
    nb = ilaenv(1, "DSYTRI", "U", n, -1, -1, -1)
    lwork = n * nb

    allocate (work(lwork))

    ! Invert the matrix.
    call dsytrf("U", n, a, n, ipiv, work, lwork, info)
    if (info /= 0) stop "error in call to dsytrf"
    call dsytri("U", n, a, n, ipiv, work, info)
    if (info /= 0) stop "error in call to dsytri"

    deallocate (work)

    ! Copy the upper triangular part of A to the lower.
    do j = 1, n - 1
        do i = j + 1, n
            a(i, j) = a(j, i)
        end do
    end do

end subroutine
```

Listing A.2: examples/calculateEigensystem.f90

```
! Calculates the eigenvalues and, optionally, the eigenvectors of a
! real symmetric matrix.
!
! jobz - If jobz = "N", compute eigenvalues and eigenvectors, if jobz = "V",
!        compute both.
! a    - On input, a real symmetric matrix. Only the upper diagonal part is
!        accessed.
! w    - On output, contains the eigenvalues of A.
! v    - On output, if jobz = "V", the columns are the eigenvectors of A.
subroutine mateigrs(jobz, a, w, v)

    implicit none

    character, intent(in) :: jobz
    real(8), intent(in)   :: a(:, :)
    real(8), intent(out)  :: w(:), v(:, :)

    integer              :: n, lwork, info
    real(8), allocatable :: work(:)

    n = size(a, 1)
    v = a

    ! Query the optimal size of the workspace.
    allocate (work(1))
    lwork = -1
    call dsyev(jobz, "U", n, v, n, w, work, lwork, info)
    lwork = int(work(1))
    deallocate (work)

    allocate (work(lwork))

    ! Diagonalize the matrix.
    call dsyev(jobz, "U", n, v, n, w, work, lwork, info)
    if (info /= 0) stop "error in call to dsyev"

    deallocate (work)

end subroutine
```

Appendix B

Plotting with PLplot

Plotting the results of your calculation from Fortran is easiest with PLplot. PLplot is a cross-platform scientific plotting package with support for multiple programming languages. Under Ubuntu, you need to install the packages `libplplot-dev` and `plplot11-driver-cairo`. The `plplot-doc` package might also come in handy. If you have it installed, you can access the documentation of, *e.g.*, the `plline` subroutine by typing

```
$ man plline
```

on the command line.

In order to successfully compile a program using PLplot, you need quite a few compiler flags and libraries. However, Ubuntu (and most other distributions) is smart enough to figure that out for you. Add the following two lines near the top of your `Makefile`, but after the declarations of `FFLAGS` and `LIBS`:

```
FFLAGS += $(shell pkg-config --cflags plplotd-f95)
LIBS += $(shell pkg-config --libs plplotd-f95)
```

The `pkg-config` command will generate the correct compiler flags on the fly.

B.1 Plotting functions

As a quick introduction to using PLplot, we'll go through Listing B.1 step by step:

Listing B.1: examples/PLplotDemo.f90

```
1  program PLplotDemo
2
3      use plplot
4
5      implicit none
6
7      call plparseopts(PL_PARSE_FULL)
```

```
8
9       ! gnuplot color scheme
10      call plscol0(0, 255, 255, 255)  ! white
11      call plscol0(1, 255, 0, 0)      ! red
12      call plscol0(2, 0, 255, 0)      ! green
13      call plscol0(3, 0, 0, 255)      ! blue
14      call plscol0(4, 255, 0, 255)    ! magenta
15      call plscol0(5, 0, 255, 255)    ! cyan
16      call plscol0(6, 255, 255, 0)    ! yellow
17      call plscol0(7, 0, 0, 0)        ! black
18      call plscol0(8, 255, 76, 0)     ! orange
19      call plscol0(9, 128, 128, 128)  ! gray
20
21      call plinit()
22
23      call plot_x2(0d0, 1d0, 21)
24
25      call plend()
26
27  contains
28
29      subroutine linspace(x1, x2, n, x)
30
31          real(8), intent(in)  :: x1, x2
32          integer, intent(in)  :: n
33          real(8), intent(out) :: x(n)
34
35          real(8) :: dx
36          integer :: i
37
38          dx = (x2 - x1) / (n - 1)
39          do i = 1, n
40              x(i) = x1 + (i - 1) * dx
41          end do
42
43      end subroutine
44
45      subroutine plot_x2(x1, x2, n)
46
47          real(8), intent(in) :: x1, x2
48          integer, intent(in) :: n
49
50          real(8) :: x(n), y(n)
51          integer :: i
```

```
52
53          call linspace(x1, x2, n, x)
54          do i = 1, n
55              y(i) = x(i)**2
56          end do
57
58          call plcol0(7)
59          call plenv(x(1), x(n), y(1), y(n), 0, 0)
60          call pllab("x", "y", "y=x#u2")
61
62          call plcol0(1)
63          call plline(x, y)
64
65          call plcol0(2)
66          call plpoin(x, y, 2)
67
68      end subroutine
69
70  end program
```

- `use plplot` imports the PLplot functions and subroutines into our program.
- `call plparseopts(PL_PARSE_FULL)` parses any command-line arguments of your program that are relevant for PLplot (see below).
- By default, PLplot plots all figures on a black background with red axes. This is inconvenient, especially for printed figures. Lines 11–19 change the color scheme of color map 0 (which is used for line plots) to that of gnuplot (with the exception of color 0, since that's the background color).
- All plotting commands must be given between the calls to `plinit` and `plend`.
- `plot_x2` plots the function $y = x^2$ for `n` values between `x1` and `x2`.
- The `linspace` subroutine generates a linearly spaced array, similar to the MATLAB function of the same name.
- `call plcol0(7)` sets the current foreground color to 7 (black), which we use for the axes and labels. The call to `plenv` sets the minimum and maximum values of the axes, scales the plot to fill the screen (the first `0` parameter) and draws a box with tick marks (the second `0`). `pllab` is then used to set the x and y labels and the title of the plot.
- Finally, we draw a red line (color 1) through the points with `plline` and draw the points themselves in green (color 2) with `plpoin`. The points are drawn with the '+' glyph (code 2).

If the program compiles successfully, you can run it with

```
$ ./plplotdemo -dev xcairo
```

The argument `-dev xcairo` tells PLplot to use the Cairo X Windows driver, which will open a window showing the plot. It is also possible to generate a PostScript image by typing

```
$ ./plplotdemo -dev pscairo -o demo.ps
```

and similarly a PDF with `pdfcairo`. You can find out about the command-line parameters by running

```
$ ./plplotdemo -h
```

or by specifying no arguments at all.

B.2 3D animations

For some programs, such as a molecular dynamics simulation, it is convenient to be able to watch the motion of particles in real time. For this we'll make use of the 3D plotting facilities of PLplot. First, we setup the viewport with

Listing B.2: examples/PLplot3D.f90

```
1  program PLplot3d
2    implicit none
3
4    call plsdev("xcairo")
5    call plinit()
6
7    call pladv(0)
8    call plvpor(0d0, 1d0, 0d0, 1d0)
9    call plwind(-1d0, 1d0, -2d0 / 3, 4d0 / 3)
10   call plw3d(1d0, 1d0, 1d0, xmin, xmax, ymin, ymax, zmin, zmax, 45d0, -45d0)
```

Since the animation will play real-time, we specify the Cairo X Windows driver directly with `plsdev` instead of parsing the command line. Lines 5–7 map the world coordinates—bounded by `xmin`, `xmax`, *etc.*—to the screen. This will result in a box which is rotated by 45 degrees around the z and x axes. We end the program with

```
11   call plspause(.false.)
12   call plend()
```

The call to `plspause` makes sure the program terminates after the animation finishes, instead of leaving the last frame on screen indefinitely.

We plot particles by calling the following subroutine:

```
14 contains
15
16   subroutine plot_points(xyz)
```

```
    real(8), intent(in) :: xyz(:, :)

    call plclear()
    call plcol0(1)
    call plbox3("bnstu", "x", 0d0, 0, "bnstu", "y", 0d0, 0, "bcnmstuv", &
                "z", 0d0, 0)
    call plcol0(2)
    call plpoin3(xyz(1, :), xyz(2, :), xyz(3, :), 4)
    call plflush()
  end subroutine

end program PLplot3d
```

`xyz` is a $3 \times n$ array, with the first row containing the x coordinates of the particles, the second the y, and the third the z coordinates. We first clear the screen with `plclear`, then draw the axis with `plbox3`, and finally the particles with `plpoin3`. The particles are drawn as small circles (code 4). Finally, we update the screen by calling `plflush`. Calling `plot_points` repeatedly results in a smooth animation. Although this works quite well for checking the movement of particles, PLplot is not a 3D engine and won't run at more than about 100 frames per second. We therefore recommend not plotting every step in your simulation, but skipping a few frames. For more documentation on PLplot, Google is your friend.

Appendix C

Parallel computing: employing a thousand monkeys

"Here be dragons."

If you have a multicore processor, or even multiple processors or machines available, one obvious way to speed up your code is to employ multiprocessing. However, this is a difficult and dangerous endeavor, especially if you don't have any prior experience. Parallel programs are far more difficult to write and debug than sequential ones, mostly because concurrency introduces entire classes of potential bugs, like race conditions and deadlocks. Also, even if it does work, it's not always beneficial. Some algorithms simply don't scale to multiple processors. Nevertheless, parallel programming is unavoidable in high-performance computing. While a true introduction is outside the scope of these notes, we'll give a few guidelines that might help you from accidentally shooting yourself in the foot:

- **Make sure your program works first.** Although this is true for optimizations in general, it is doubly so for parallel programs. Programs are much harder to debug when they run in parallel, so leave parallelization for the final step.
- **Check whether the bottleneck in your program benefits from parallelization.** Not every algorithm scales well. It would be a waste of time to spend effort parallelizing your code only to discover that it's now actually slower.
- **Use OpenMP.**[1] For ICCP you won't be using supercomputers or systems with multiple nodes. Sophisticated libraries like MPI[2] are overkill and introduce needless complexity. OpenMP only works on single machines with shared memory, but it is by far the easiest framework to use. All you need to do is add `-fopenmp` to `FFLAGS` and `LDFLAGS` in your `Makefile` and add a few directives to your code. For example

[1] OpenMP is available for C, C++and Fortran. If you are using C++, the *Intel Threading Building Blocks* library is a possible alternative; it has high-level support for certain algorithms and data structures.

[2] Message Passing Interface: a standardized protocol for communications between nodes in a cluster—used on pretty much every supercomputer in the TOP500.

```
!$omp parallel do
do i = 1, n
    y(i) = x(i) * i
end do
!$omp end parallel do
```

will execute the loop in parallel. Use Google for more information on how to use OpenMP.

- **Don't use more threads than you have physical cores.** This may seem like stating the obvious, but it actually happens rather often. Most modern Intel processors implement hyper-threading. This doubles the apparent number of cores, which helps with task switching when you're running many different programs at once. However, for CPU intensive calculations, like the ones you'll be running, the threads will be competing for the available physical cores. In that case doubling the number of threads will actually harm performance.

Appendix D

Revision control

> *"And then there's `git rebase -interactive`, which is a bit like `git commit -amend` hopped up on acid and holding a chainsaw—completely insane and quite dangerous but capable of exposing entirely new states of mind."*
>
> —Ryan Tomayko

A revision control system is perhaps the third most useful tool to a programmer after only the editor and compiler. The idea is simple: you give a piece of software a description of recent changes you've made to your code, and that software will efficiently log the changes and description for you to view and optionally undo should you need to. Written by Linus Torvalds and popularized by websites like Github, Git is probably the most widely used revision system around. While a a full Git tutorial is far beyond the scope of this introduction, knowledge of a few basic commands can potentially save you a ton of time (and headaches!) down the road.

To get started, you'll want to install the `git`, `gitk`, and `git-gui` packages on Ubuntu. Next, let's assume you've created an `iccp/` directory with one file, `myProg.f90`, that already has some content. Navigate to the `iccp/` directory and type

```
$ git init
$ git add .
$ git commit -m ‘‘Initial commit’’
```

This does a few things for you:

- `git init` creates a git repository of `iccp/` by placing the current directory (and all subdirectories) under revision control. Any files and directories that get added to `iccp/` can be tracked as part of the current project.
- `git add .` adds every file's local changes to the *staging area* or *index*. The index tracks changes for you to inspect before they become an official part of the project history.
- `git commit -m “Initial commit”` takes everything in the index and adds it to the history with the description "Initial commit".

You can add future changes in `myProg.f90` to the history in a similar way:

```
$ git add myProg.f90              # or . to add everything
$ git commit -m ''Fixed bug XYZ'' # use a msg appropriate for your changes
```

Similarly, you can check the current status of the index with `git status`. The output will display changes both staged and not staged in the index for the next commit, allowing you to tweak things as necessary.

If, after some bleary-eyed, hungover coding, you realize you've made a horrible mistake since your last commit, you can check a file's current state against the most recent history with

```
$ git diff <file>
```

And to delete any changes you've made

```
$ git checkout <file>
```

You can view the entire project history with the `gitk` program, including file diffs, commit messages, and authors. Similarly, `git gui` provides a handy interface for staging commits. You can write/amend commit messages, see which files have changed and what files you've added to the index, push/pull to remote repositories, etc.

There are only a handful of ways to completely delete data from the repository once you've committed it[1], so unless you're *really* trying to rewrite history your code is reasonably safe from git-mishaps. That said, here are some tried and true tips to make the most of Git:

- **Commit frequently**. The more commits you have, the more the history can help you. If you're trying to find a problem introduced in a new subroutine, it's much easier to search through ten lines of code you added in the past half hour than the 200 lines you added since Monday.

- **Commit *logically***. Commit complete logical units (like writing a function, fixing a bug, or formatting output) instead of just "at the end of the day." This greatly helps in reasoning about the development of your code—and thus undoing problems!

- **Use descriptive commit messages**. Messages like "Fixed bug in force calculation" and "Implemented averaging function" work far better than an endless series of "nightly backup"s. If you can't easily reason about what's changed in your code, tracking the changes will do very little to help you.

- **Commit the bare minimum**. Only commit what's needed to recompile a functional program. Git just tracks the differences in files between commits to save space, but executables, output data, PDFs, images, etc. all require a totally new copy in the history every time they change (and that's frequently!) Tracking a `.gitignore` file with some simple patterns can automatically ignore files of a similar type.

[1] It's a good idea to avoid the more dangerous commands like `git rebase` and `git gc`. *Always* think about commands before you run them!

- **Learn the advanced Git features**. Basic commits barely scratch the surface of Git's functionality. Branching lets you test drive changes in a copy of your code in an easily-reset sandbox, the network features let you not only backup code but also collaborate with other developers, and `git-bisect` can help you *automatically* pinpoint exactly where and when a bug crept in.

You can find all of the example codes given in these notes at the libICCP repository on Github. To use them, you'll want to `clone`[2] the repository with

```
$ git clone git@github.com:cglosser/libICCP.git
```

You can then download any future changes to the repository with a `git pull` command inside the libICCP directory

Because of Git's popularity, the internet has a number of fantastic resources for learning to use it. Github Learning, Pro Git, Think Like a Git, and Git Ready all offer very good tutorials and explanations.

[2] A clone is a complete local backup of the repository. It contains the entire project history and may itself be cloned and pulled from.

Further reading

a.k.a programmer humor

- Real programmers don't use Pascal (**Mandatory**)
- The many faces of function definition in Fortran 95 (**Mandatory**)
- The Ten Commandments of Egoless Programming
- A Brief, Incomplete, and Mostly Wrong History of Programming
- `Ed, man!  !man ed`
- Vim koans